Binary Search Tree (BST)

Definition

- A **Binary Search Tree (BST)** is a binary tree where each node has at most two children, and the left child is smaller than the parent, while the right child is greater.

Properties

- Left subtree nodes < Root < Right subtree nodes
- In-order traversal yields elements in sorted order
- Average Time Complexity:
    - **Search**: O(log n)
    - **Insertion**: O(log n)
    - **Deletion**: O(log n)

Operations

1. **Search**: Compare the target with the root, recurse into left or right subtree accordingly.
2. **Insertion**: Recursively find the correct position and insert.
3. **Deletion**:
    - Case 1: Node is a leaf → Simply remove it.
    - Case 2: Node has one child → Replace node with its child.
    - Case 3: Node has two children → Replace with in-order successor (smallest node in right subtree).

Solving Insertion Problems

To insert a value **val** into a BST:

1. If the tree is empty, create a new node as the root.
2. Compare **val** with the root:
    - If **val** is smaller, recursively insert into the left subtree.
    - If **val** is greater, recursively insert into the right subtree.
3. Return the root after insertion.

Code Example:

```
class TreeNode:
    def __init__(self, key):
        self.val = key
        self.left = None
```

```
        self.right = None

    def insert(root, key):
        if root is None:
            return TreeNode(key)
        if key < root.val:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
        return root
```

AVL Tree

Definition

- An **AVL Tree** is a self-balancing BST where the height difference (balance factor) of left and right subtrees of any node is at most **1**.

Properties

- Balance Factor = **height(left subtree) - height(right subtree)**
- Ensures O(log n) operations

Rotations (to maintain balance)

1. **Right Rotation (LL case)**
2. **Left Rotation (RR case)**
3. **Left-Right Rotation (LR case)**
4. **Right-Left Rotation (RL case)**

Operations

- **Insertion**: Insert like in BST, then rotate if unbalanced.
- **Deletion**: Delete like in BST, then balance.
- **Search**: Same as BST (O(log n)).

Solving Insertion Problems

To insert a value **val** into an AVL Tree:

1. Insert the value like in a BST.
2. Update balance factors and check for violations.
3. Perform necessary rotations (LL, RR, LR, RL) to restore balance.

Code Example:

```
class AVLNode:
    def __init__(self, key):
        self.val = key
        self.left = None
        self.right = None
        self.height = 1

def insert(root, key):
    if not root:
        return AVLNode(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

    root.height = 1 + max(get_height(root.left), get_height(root.right))
    balance = get_balance(root)

    if balance > 1 and key < root.left.val:
        return rotate_right(root)
    if balance < -1 and key > root.right.val:
        return rotate_left(root)
    if balance > 1 and key > root.left.val:
        root.left = rotate_left(root.left)
        return rotate_right(root)
    if balance < -1 and key < root.right.val:
        root.right = rotate_right(root.right)
        return rotate_left(root)

    return root
```

Hashmap

Definition

- A **Hashmap** (or Hash Table) is a data structure that maps keys to values using a hash function.

Properties

- Uses a **hash function** to compute an index (hash) for keys.
- Supports **average O(1) time complexity** for insert, delete, and search.

Collision Handling Methods

1. **Chaining**: Use a linked list at each bucket.
2. **Open Addressing**:
    - **Linear Probing**: Search next available slot.
    - **Quadratic Probing**: Use a quadratic function to find a slot.
    - **Double Hashing**: Use another hash function for collision resolution.

Operations

- **Insertion**: Compute hash, place in bucket (handle collisions if necessary).
- **Search**: Compute hash, check bucket.
- **Deletion**: Find key and remove it.

Solving Insertion Problems

To insert a key-value pair **(key, value)** into a hashmap:

1. Compute the hash of **key**.
2. Check if the bucket is empty; if so, insert.
3. If occupied, resolve collision (chaining or probing).

Code Example:

```
class HashMap:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
```

```
    return hash(key) % self.size

def insert(self, key, value):
    index = self._hash(key)
    for pair in self.table[index]:
        if pair[0] == key:
            pair[1] = value
            return
    self.table[index].append([key, value])
```